02｜3KU法则：如何找出最优自动化实施截面？

2022-03-23 柳胜

《自动化测试高手课》 课程介绍 >

讲述：柳胜

时长 17:55 大小 16.42M

你好，我是柳胜。

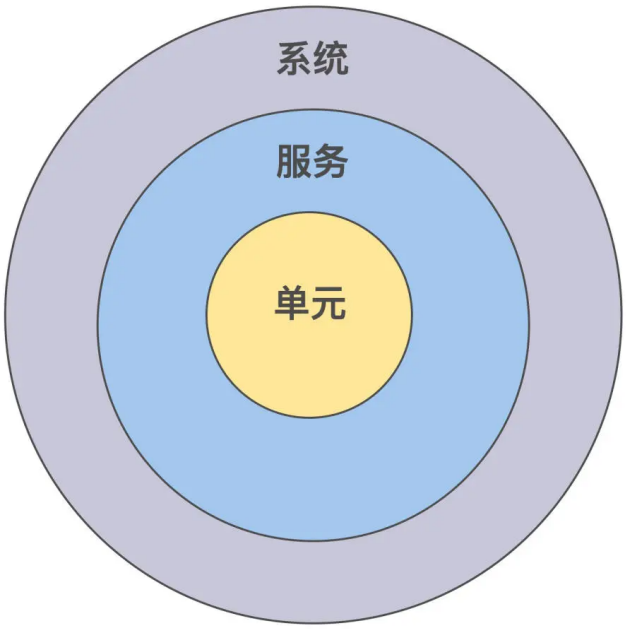
上一讲我们提出了自动化测试 ROI 模型，在回归测试中的应用。回归测试是一个笼统的概念，单元测试、接口测试以及 UI 测试里都有回归测试，甚至性能测试也已经成为回归测试的一部分。

今天我们要关注一个具体场景，给你一个软件系统，作为自动化测试人员，你怎么找出测试截面，制定自动化测试方案？这些事可能你都做过，觉得并不稀奇，但既然我们已经学习了 ROI 思维，今天要再加上一个小目标，**制定策略，能够让这个自动化测试设计获得可能大的 ROI**。换句话说，能干还不够，还要干得好，既要马儿跑，又要马儿少吃草。

有挑战不？那就跟我进入这一讲的学习，一起找到最佳策略吧。

测试 ROI 金字塔

在测试设计领域，经常提到的方法是分层。具体就是给定一个系统，结构上划分三个层级，单元在最小圈；服务包含多个单元，在中圈；而系统又包含多个服务，是外部的最大圈。结构图如下：

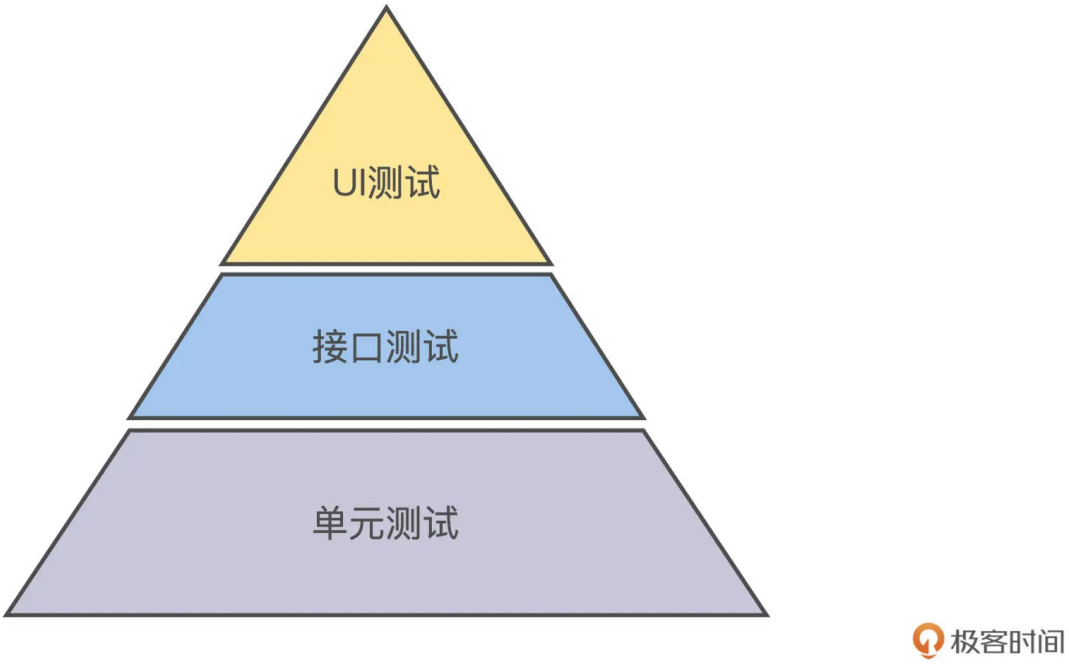


极客时间

软件结构圈图

相应地，我们的测试结构是在代码层做单元测试，服务层做接口测试，系统层做 UI 功能测试。

在实践中，这三种测试该怎么组合安排呢？迈克·科恩在 2009 年他的新书《敏捷成功之道》中首次提出了测试金字塔模型。单元测试自动化在金字塔底部，接口测试自动化在中部，而 UI 测试自动化在金字塔顶部。



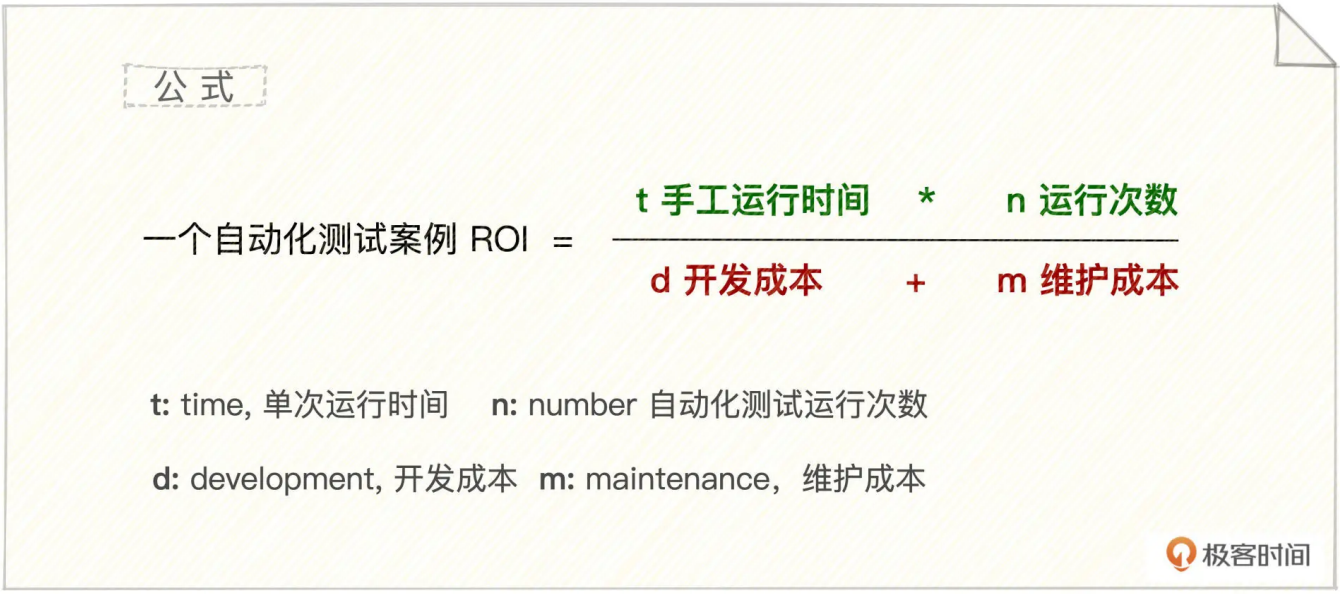
分层测试金字塔

迈克·科恩讲到自动化测试工作量配比时，认为应该按照层面积分配。也就是说，单元测试案例的数目应该多于接口测试案例数目，接口测试案例数目应该多于 UI 测试自动化测试案例数目。

后来，金字塔模型又被业界发展，赋予了不同的测试策略，比如自底向上执行速度减慢，自顶向下业务属性减弱，技术属性增强。

但迈克·科恩没有解释，为什么各层工作量配比要按照测试金字塔分布？按照软件结构图，系统在最大圈，测试案例应该最多，而到了自动化测试金字塔，UI 自动化测试案例却最少；单元测试在小圈，测试案例应该最少，但到了自动化测试金字塔，单元测试案例却最多。

为什么是金字塔？要是不去理解规律背后这个“为什么”，你就用不好这个规律。上一讲我们知道了“ROI 其实是自动化测试的隐式命脉”，现在我们就利用 ROI 思维，分析一下测试金字塔规律。



ROI公式

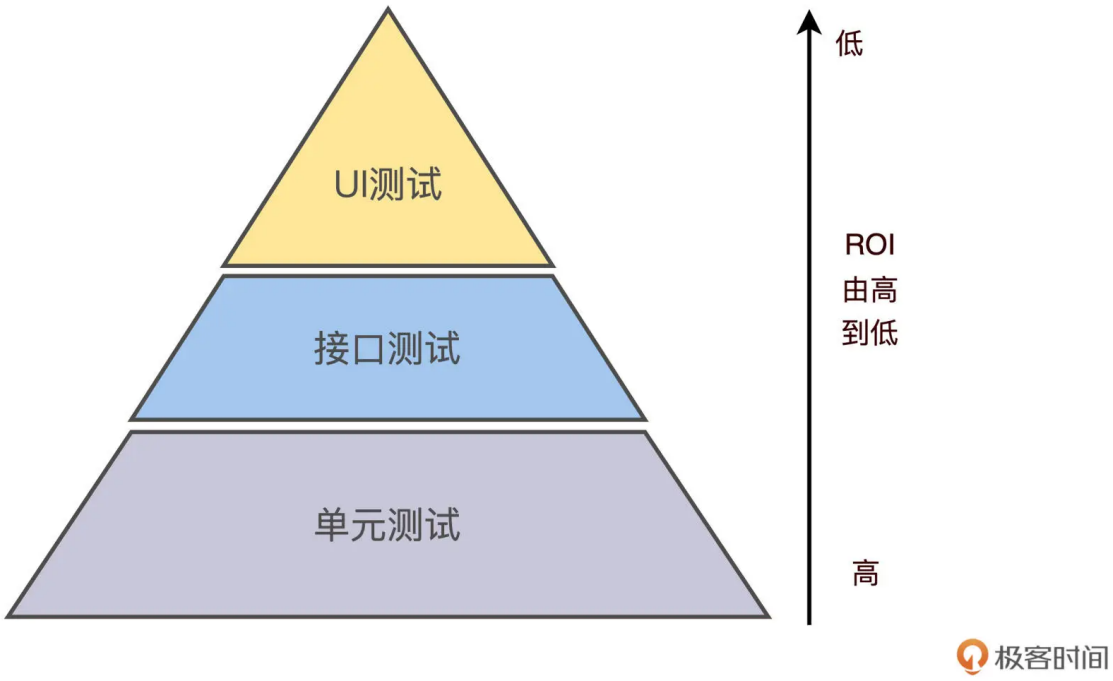
下面，我们分别看看每层的 ROI。单元测试可以在开发人员每次 code commit 触发运行，回归频率高；接口测试在每轮集成测试运行，回归频率中；UI 自动化测试在用户验收测试，回归频率低。

按照 ROI 模型，我们可以得出 3 种类型自动化测试的 ROI 排序，如下表：

测试类型	t	n	d	m	ROI
单元测试	无法手工	次数多	低	低	高
接口测试	中	次数中	中	中	中
UI自动化测试	长	次数少	中	非常高	低

极客时间

对照测试金字塔不难发现，实际上三类自动化测试的 ROI 是自底向上由高到低的。



分层测试ROI金字塔

按照第一讲得出的规律“自动化测试顺序从 ROI 高到低”，我们优先投入精力做 ROI 最高的单元测试，再做 ROI 中的接口测试，最后完成 UI 测试。

现在就可以轻松解释迈克·科恩的金字塔了，因为 ROI 存在差异，所以按照高 ROI 大投入，中 ROI 中投入，低 ROI 小投入，工作量比例呈金字塔分布，底层面积最大，顶层面积最小。发现没？**根源在于 ROI，金字塔是表现出来的形态而已。**

好，到这里，总结一下。各种软件理论学派，大致可以分为两种，一种是理论基础，讲的是做什么，比如软件测试定义、软件过程，另外一种是实践经验，讲的是该怎么做，比如金字塔模型。

实践和理论很大的不同就是在现实商业中，我们不可能完全按照理想来工作，而是要加入很多制约因素，其中最大的制约就是钱。明白这个道理，你就会知道为什么 ROI 是根源，你也会知道怎么能够在工作中做出业绩了，不是耍两个工具，忽悠一下领导就算成功，而是认认真真地去思考，踏踏实实地去提高 ROI，直到边际效应 ROI 无法提高为止。

寻找最优 ROI 策略

刚才说了分层测试和各层 ROI，业界也很认可这种分层理论，但实际落地时却存在问题：一批人做 UI 测试自动化，另外一批人去做接口测试，然后开发人员做单元测试。三路人马忙得不亦乐乎，都说自己贡献大，等到 bug 发生了泄漏到生产环境，又开始甩锅。

分层测试为啥会“内卷”

很明显，这是一个内卷的场景，让我们结合例子具体看看内卷发生在哪里？

以一个 Web 登录操作为例，用户在 UI 上输入用户名和密码，点击“登录”按钮。Selenium UI 自动化会这样实现：

复制代码

```
@Test
public void login() {
  WebDriver driver=new ChromeDriver();
  driver.manage().window().maximize();
  //打开页面
  driver.get("https://www.example.com/users/sign_in");
  WebElement username=driver.findElement(By.id("user_email_Login"));
  WebElement password=driver.findElement(By.id("user_password"));
  WebElement login=driver.findElement(By.name("login"));
  //输入用户名
  username.sendKeys("liusheng@example.com");
  //输入密码
  password.sendKeys("123456");
  //点击登录按钮
  login.click();
}
```

上面 UI 的操作被 Web 服务转化成 Rest 请求，进入到 API 网关，是这样的：

复制代码

```
curl --location --request POST 'http://auth.example.com/auth/realms/Test/users
--header 'Content-Type: application/x-www-form-urlencoded' \
--header 'username=liusheng@example.com' \
--header 'password=123456'
```

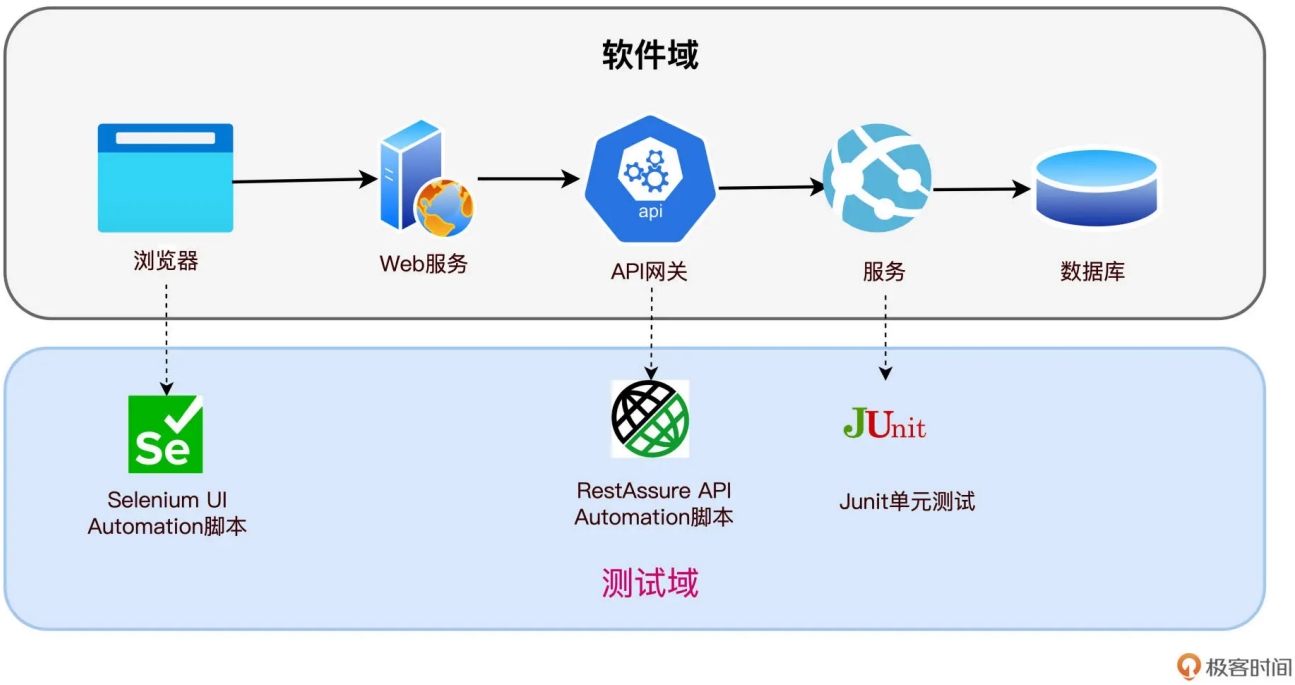
在单元上执行的则是这样的代码：

复制代码

```
public Future<ResponseData> login(String userName, String password) {
    //入口参数检验
    if (StringUtil.isBlank(userName)||StringUtil.isBlank(password)){
      return new AsyncResult<>(ResponseData.error("账号密码不能为空"));
    }
    //查询用户是否存在
```

```
    List<User> userList = baseMapper.getUserInfo(userName);
    if (CollectionUtils.isEmpty(userList)){
        return new AsyncResult<>(ResponseData.error("账号不存在"));
    }
    //验证账号密码是否正确
    User user = userList.get(0);
    String requestMd5 = SaltUtil.md5Encrypt(password, user.getSalt());
    String dbMd5 = user.getPassword();
    if (dbMd5 == null || !dbMd5.equalsIgnoreCase(requestMd5)) {
        return new AsyncResult<>(ResponseData.error("账户密码不正确"));
    }
    //生成 access token，并返回
    String token = JwtTokenUtil.generateToken(user);
    return new (ResponseData.success(token));
}
```

可以看到，一个请求，从浏览器页面发起，进入 API 网关，再传递到服务里的 Login 函数，经过了 UI 测试、API 测试和单元测试三个测试截面。



三个测试截面示意图

三个测试截面测的是一个请求在不同层面上的形态，那么每一个截面都可以测试全部的案例，也可以测试部分的案例。就像 3 个人负责 1 个项目一样，如果没有经过事先的协调和安排，3 个人可能做了重复的事情，造成浪费，也可能存在一件事 3 个人都没干，形成测试盲区。

需求 / 策略矩阵

这种“内卷”是不是一个问题？可能你会说没问题，各层独立测试能够加强质量保障。说这话的底气在于测试上的投入充足，不计内卷成本。实际上，在 DevOps 风行的今天，趋势是追求效果和效率。所以，在资源有限的条件下，我们需要在整体上看待分层测试的最优 ROI。

咱们先看看测试需求是什么，用 FURPS 模型来理一下需求。FURPS 是用 5 个维度来描述一个软件的功能需求，FURPS 这个单词对应着每个需求的英文首字母：

F=Function 功能

U=Usability 易用性

R=Reliability 可靠性

P=Performance 性能

S=Supportability 可支持性

把测试需求和测试类型组合在一起，就整合了后面这个矩阵表格：

测试需求/测试类型	功能			易用性	可靠性	性能	可支持性
	场景	数据	算法				
UI测试	可以	可以	可以	可以	可以	可以	可以
接口测试	部分	可以	可以	不	可以	可以	部分
单元测试	不	部分	可以	不	部分	可以	部分

极客时间

结合表格，可以看到 UI 测试、接口测试和单元测试每个截面的测试能力。

在 UI 层面上，功能性最强，所有测试需求都可以做。这个可以理解，因为软件本身就是满足用户需求，没有一个需求不可以从用户层面感受到。如果真的存在一个需求，用户却无法体验到，那根据奥卡姆剃须刀原理，这种用户无法体验到的需求就是无效的。

接口层面上，功能性减弱，技术性增强。

单元层面上，技术性最强，功能性主要体现在数据的处理，算法逻辑上。

3KU 整体策略

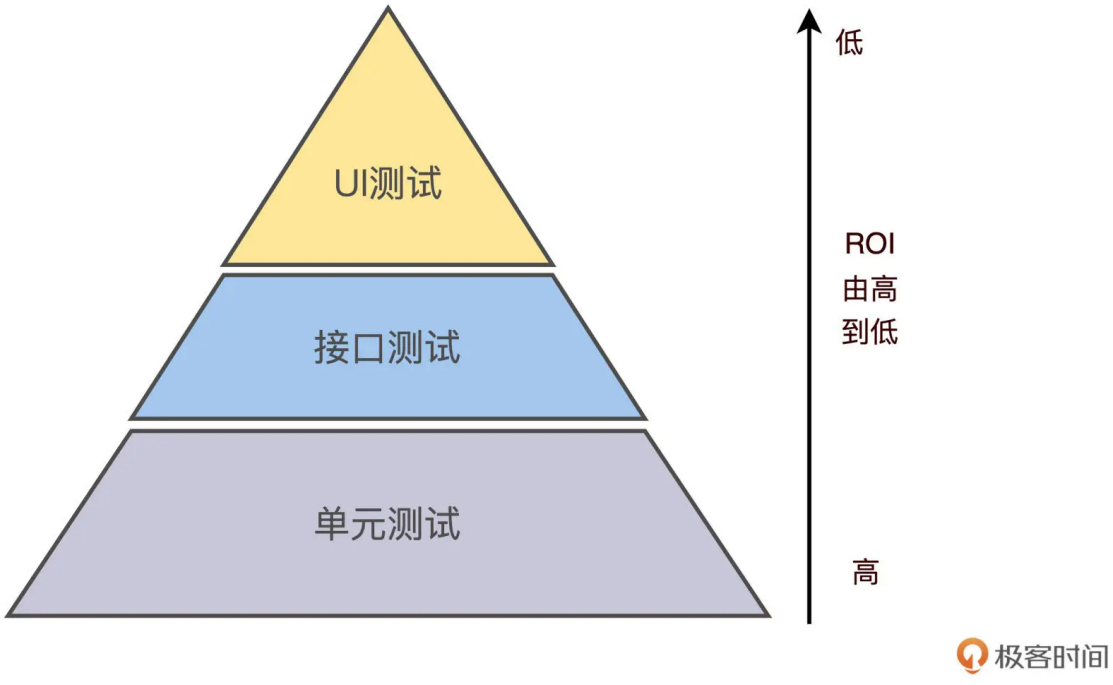
好，有了需求 / 策略矩阵后，结合上面讲到的自动化测试 ROI 金字塔，我们的整体最优 ROI 策略就呼之欲出了。什么是整体最优 ROI 呢？

有 3 个 Key（关键因素）：

Useful: 每个测试需求都是有效的；

Ultimate: 每个测试需求的验证都在优先寻找自动化 ROI 高的层面去实现，如果不可行，按照 ROI 高到低回退，直到 UI 层；

Unique: 每个层面上验证的测试需求都和别的层面都不是重复的。
这样分配的工作，既不重复，又没遗漏，还遵循了 ROI 的原则。我管它叫 **3KU 原则。**



3KU测试金字塔

3KU 策略该怎么执行呢? 按照 3KU 策略，我们把表格里的测试需求，对照下面这三个问题，按顺序检查一遍：

1. 能在单元测试验证么？

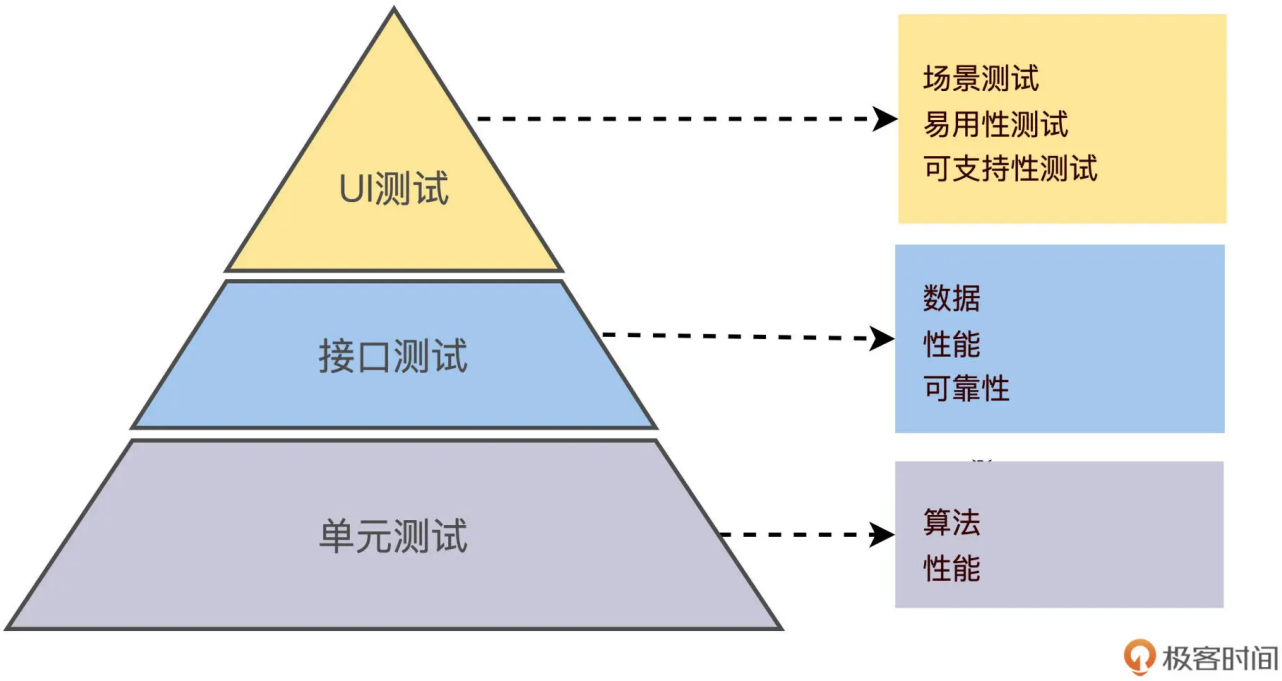
2. 能在接口测试验证么？

3. 能在 UI 测试验证么？

这样检查以后，就能得出各个需求的自动化实现截面了。

UI 测试关注功能场景测试，易用性测试和可执行性测试；而接口测试关注不同数据的循环，接口的性能和错误恢复能力；单元测试关注算法的正确性和性能。

恭喜你看到这里，最后就是我们收割成果的环节了。我们又得出了一个满足 3KU 原则的自动化测试实施金字塔，各层有自己的关注点，又在整体上实现了互相配合补偿。



3KU测试金字塔

在 3KU 测试金字塔下，每一个测试需求都会选择最大的 ROI 测试截面，通过这样的安排，实现了整体最优 ROI 的目标。对不对？

小结

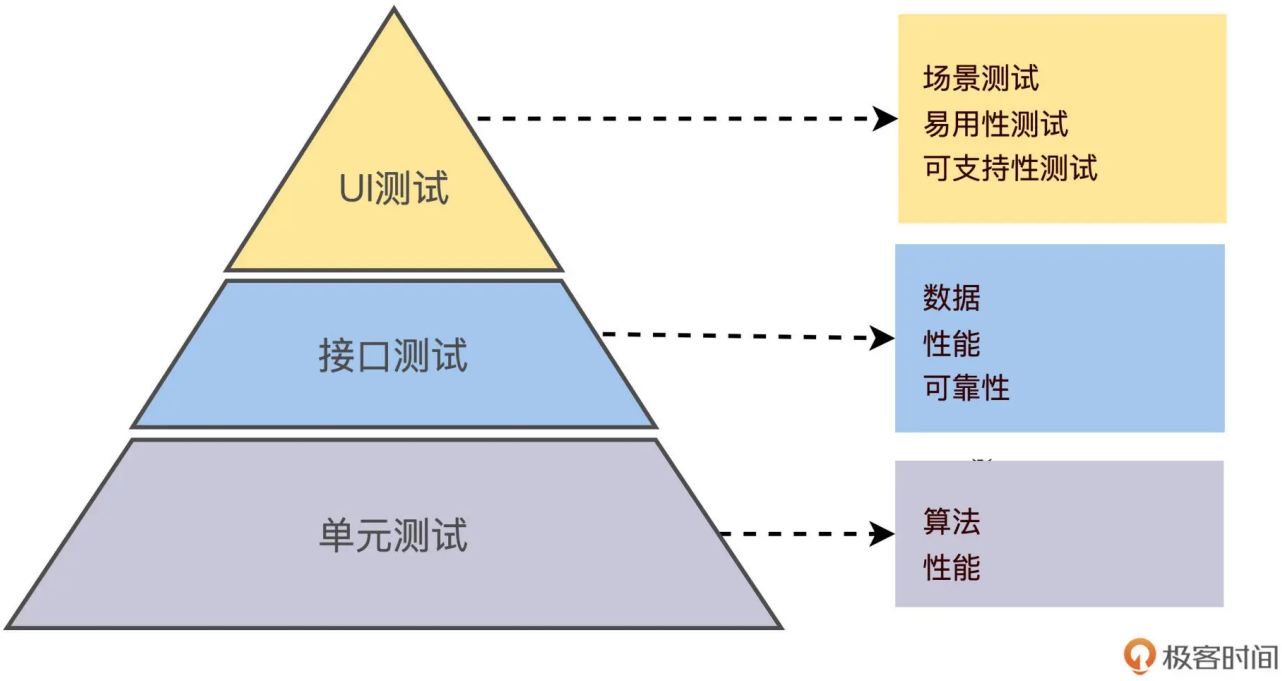
这一讲，我们从 ROI 角度分析了一下分层测试的原理和在实践中的应用。先入为主地，分层理论上的分层测试的特性，必然会造成重叠和错失。这给测试从业者带来了挑战。但挑战也是机会，如何解决这个问题？

这就需要我们遵循回归到效益的原则，思考怎么用最少的资源干最多的事，能达到这个效果，就是好的实践。因此，我们提出了分层但协调实现整体最优 ROI 的解决方案，3KU 测试矩阵和 3KU 测试金字塔。

测试需求/测试类型	功能			易用性	可靠性	性能	可支持性
	场景	数据	算法				
UI测试	可以	可以	可以	可以	可以	可以	可以
接口测试	部分	可以	可以	不	可以	可以	部分
单元测试	不	部分	可以	不	部分	可以	部分

极客时间

3KU测试矩阵



3KU测试金字塔

沿着这个思路，各层做好自己具有优势能力的测试需求，比起全部需求系于端到端的测试上，更有效率和效益，**分层是追求整体 ROI 的结果**。之后的课程里我们还会反复提到 ROI，最后你也会不由感叹，ROI 是背后无形的大手，大道无形，无处不在。

思考题

1 软件大师马丁·福勒曾经说过：“在微服务时代，分层测试不再呈现金字塔形状。”这是为什么？试着用 ROI 来解释一下。

2 学完今天的内容，如果你是测试主管，你希望你的团队是全栈（一个人负责一个模块的所有层面测试），还是精细分工（一个人负责所有模块的一个层面测试）？有什么优劣？

欢迎你在留言区跟我交流互动，如果这一讲对你有启发，也推荐你分享给身边更多同事和朋友。

分享给需要的人，Ta订阅超级会员，你最高得 50 元

Ta单独购买本课程，你将得 20 元

生成海报并分享

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 01 | ROI价值内核：自动化测试的价值可以量化么？

下一篇 03 | 工具选择：什么工具框架值得用？

精选留言 (10)

写留言

D_w

2022-03-23

一点浅见： 问题1：在微服务架构下，测试层级会扩充，新增契约测试，组件测试，端到端的测试等等，并且在实际测试任务中一般开发负责的单元测试占比不高，最终成纺锤形。 问题2：如果团队成员能力都比较强，我偏向全栈，如果能力不均衡还是得考虑精细分工。我觉得全栈的测试效果能更好

展开

作者回复: 我在甲骨文和其他公司经历过几轮调整，每一轮过后，专职系统测试人员就会减少一部分，这个背后的逻辑就是测试下沉，单元测试和API测试做强，有了这个基础后，端到端测试做轻巧。被变化，不如准备好，拥抱变化！

1

太匆匆
2022-03-23

我们很多人都喜欢引用金字塔模型，但是似乎没有多少人认真思考过金字塔的模型是怎么来的，有无理论依据。这点得向老师学习，拒绝拿来主义

作者回复: 你有这个问号，已经很有悟性了！

1

lisa
2022-03-26

我希望是最重视全栈的模式，因为对技术的理解成本远远小于对业务的理解成本，也就是一个人把它培训成全栈可能花上一个月就行，但是让这个人充分理解业务以及业务的架构是一件比较长期的事情

作者回复: 你能从成本的角度看待问题了，大大的👍！我是用了好几年才悟到这些！

lisa
2022-03-26

微服务本身基于RPC调用，服务拆分的力度比较小，所有从ROI的模型来讲，微服务的代码每次commit以及merge/pull request会进行API测试，这样API测试的运行次数n会加大，分子变大。而相对于单元测试来讲，API测试的建设成本d和维护成本m相对于单元测试会变小很多，导致分母变小。所以在微服务架构下，投资API测试的ROI明显更高，所以就会出现棱形更由于金字塔的情况出现。 但是说实话，我觉得说到底还是模块拆分力度变小带...

展开

swordman
2022-03-26

说说我这几年的项目经历： 1. Android移动端：全部使用ROI最高的UI自动化测试，开发和维护成本较高，虽然和CI流水线进行了集成，但由于测试用例数量的缺乏，及稳定性不足，并未达到预期的结果； 2. 微信小程序：由于业务的实时性需求，小程序前端承担了大部分的业务和算法处理。因此采用ROI最高的单元测试 + 基本场景的手工UI测试，再加上和代码提交流水线结合，取得了很好的效果； 3. C++多组件项目：正准备着手进行的项...

展开

作者回复: 你是自动化测试有经验的老兵！非常好的经验和收获体会，谢谢分享！可能有一个Typo，“ROI最高的UI自动化测试”我猜你想说的是”ROI最低的UI自动化测试“，对吧。😄

IT蜗壳-Tango

2022-03-23

思考题： 我选择全栈，因为对于特定模块来说，我可以从单元测试，API，UI三个层面去验证这个模块的性能以及可用性。 问题： 目前的自动化测试感觉很少会接触到单元测试层面，尤其是黑合测试的场景。是不是主要将UI, API的ROI考虑的全面一些就可以了啊。

展开

作者回复: 是的！ 国内不重视单元测试，可以先把ROI思维用在UI和API测试上，逐步推动单元测试。最终一定会行得通，因为没有哪个老板不喜欢ROI。

追风筝的人

2022-03-23

2. 精细分工

追风筝的人

2022-03-23

UI 测试关注功能场景测试，易用性测试和可执行性测试；而接口测试关注不同数据的循环，接口的性能和错误恢复能力；单元测试关注算法的正确性和性能。

作者回复: 是的，这是一个general的规则，可以根据软件项目实际情况进行调整。

太匆匆

2022-03-23

问题2，我仍偏向于精细分工，一人负责所有会出现效率问题，同时个人有其局限性

作者回复: 精细分工的优势是专而快，劣势是协作沟通的成本。也要相信团队，不妨让他们放手干一下！

清风明月

2022-03-23

1，我所经历的几个项目，在微服务架构，主要还是接口自动化测试，基本上不做UI自动化测试了（只做少量的app自动化测试），单元测试基本上形同虚设，主要还是做大量的接口测试！ 2，全栈的测试人员很难招到，而且现在微服务测试人员最好是熟悉所有模块，只测试局部模块意义不大，之前带团队的时候，手下有10多人，基本上大家都负责有自己的项目，所以必须熟悉整个项目，必须会做接口自动化测试！所以我还是比较倾向于：...

展开

作者回复: 是的，你的UI测试的实践很务实，可惜的是单元测试这么高的ROI测试层面。

共 2 条评论